Theano: A CPU and GPU Math Compiler in Python

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, Yoshua Bengio

Abstract—*Theano is a compiler for mathematical expressions in Python that combines the convenience of NumPy's syntax with the speed of optimized native machine language. The user composes mathematical expressions in a high-level description that mimics NumPy's syntax and semantics, while being statically typed and functional (as opposed to imperative). These expressions allow Theano to provide symbolic differentiation. Before performing computation, Theano optimizes the choice of expressions, translates them into C++ (or CUDA for GPU), compiles them into dynamically loaded Python modules, all automatically. Common machine learning algorithms implemented with Theano are from $1.6\times$ to $7.5\times$ faster than competitive alternatives (including those implemented with C/C++, NumPy/SciPy and MATLAB) when compiled for the CPU and between $6.5\times$ and $44\times$ faster when compiled for the GPU. This paper illustrates how to use Theano, outlines the scope of the compiler, provides benchmarks on both CPU and GPU processors, and explains its overall design.*

Introduction

Python is a powerful and flexible language for describing large-scale mathematical calculations, but the Python interpreter is in many cases a poor engine for executing them. One reason is that Python uses full-fledged Python objects on the heap to represent simple numeric scalars. To reduce the overhead in numeric calculations, it is important to use array types such as NumPy's `ndarray` so that single Python objects on the heap can stand for multidimensional arrays of numeric scalars, each stored efficiently in the host processor's native format.

[NumPy] provides an N-dimensional array data type, and many functions for indexing, reshaping, and performing elementary computations (`exp`, `log`, `sin`, etc.) on entire arrays at once. These functions are implemented in C for use within Python programs. However, the composition of many such NumPy functions can be unnecessarily slow when each call is dominated by the cost of transferring memory rather than the cost of performing calculations [Alted]. [numexpr] goes one step further by providing a loop fusion optimization that can glue several element-wise computations together. Unfortunately, numexpr requires an unusual syntax (the expression must be encoded as a string within the code), and at the time of this writing, numexpr is limited to optimizing element-wise computations. [Cython] and [scipy.weave] address Python's performance issue by offering a simple way to hand-write crucial segments of code in C (or a dialect of Python which can be easily compiled to C, in Cython's case). While this approach can yield significant speed gains, it is labor-intensive: if the bottleneck of a program is a large mathematical expression comprising hundreds of elementary operations, manual program optimization can be time-consuming and error-prone, making an automated approach to performance optimization highly desirable.

Theano, on the other hand, works on a symbolic representation of mathematical expressions, provided by the user in a NumPy-like syntax. Access to the full computational graph of an expression opens the door to advanced features such as symbolic differentiation of complex expressions, but more importantly allows Theano to perform local graph transformations that can correct many unnecessary, slow or numerically unstable expression patterns. Once optimized, the same graph can be used to generate CPU as well as GPU implementations (the latter using CUDA) without requiring changes to user code.

Theano is similar to [SymPy], in that both libraries manipulate symbolic mathematical graphs, but the two projects have a distinctly different focus. While SymPy implements a richer set of mathematical operations of the kind expected in a modern computer algebra system, Theano focuses on fast, efficient evaluation of primarily array-valued expressions.

Theano is free open source software, licensed under the New (3-clause) BSD license. It depends upon NumPy, and can optionally use SciPy. Theano includes many custom C and CUDA code generators which are able to specialize for particular types, sizes, and shapes of inputs; leveraging these code generators requires gcc (CPU) and nvcc (GPU) compilers, respectively. Theano can be extended with custom graph expressions, which can leverage `scipy.weave`, PyCUDA, Cython, and other numerical libraries and compilation technologies at the user's discretion. Theano has been actively and continuously developed and used since January 2008. It has been used in the preparation of numerous scientific papers and as a teaching platform for machine learning in graduate courses at l'Université de Montréal. Documentation and installation instructions can be found on Theano's website [theano]. All Theano users should subscribe to the announce[1] mailing list (low traffic). There are medium traffic mailing lists for developer discussion[2] and user support[3].

This paper is divided as follows: Case Study: Logistic Regression shows how Theano can be used to solve a simple problem in statistical prediction. Benchmarking Results presents some results of performance benchmarking on problems related to machine learning and expression evaluation. What's in Theano gives an overview of the design of Theano. Limitations and Future Work outlines current limitations of our implementation and currently planned additions to Theano.

The corresponding author is with Université de Montréal, e-mail: james.bergstra@umontreal.ca.

[1] http://groups.google.com/group/theano-announce
[2] http://groups.google.com/group/theano-dev
[3] http://groups.google.com/group/theano-users

Case Study: Logistic Regression

To get a sense of how Theano feels from a user's perspective, we will look at how to solve a binary logistic regression problem. Binary logistic regression is a classification model parameterized by a weight matrix W and bias vector b. The model estimates the probability $P(Y = 1|x)$ (which we will denote with shorthand p) that the input x belongs to class $y = 1$ as:

$$P(Y = 1|x^{(i)}) = p^{(i)} = \frac{e^{Wx^{(i)}+b}}{1 + e^{Wx^{(i)}+b}} \qquad (1)$$

The goal is to optimize the log probability of N training examples, $\mathcal{D} = \{(x^{(i)}, y^{(i)}), 0 < i \leq N\})$, with respect to W and b. To maximize the log likelihood we will instead minimize the (average) negative log likelihood[4]:

$$\ell(W,b) = -\frac{1}{N}\sum_i y^{(i)}\log p^{(i)} + (1 - y^{(i)})\log(1 - p^{(i)}) \qquad (2)$$

To make it a bit more interesting, we can also include an ℓ_2 penalty on W, giving a cost function $E(W,b)$ defined as:

$$E(W,b) = \ell(W,b) + 0.01\sum_i\sum_j w_{ij}^2 \qquad (3)$$

In this example, tuning parameters W and b will be done through stochastic gradient descent (SGD) on $E(W,b)$. Stochastic gradient descent is a method for minimizing a differentiable loss function which is the expectation of some per-example loss over a set of training examples. SGD estimates this expectation with an average over one or several examples and performs a step in the approximate direction of steepest descent. Though more sophisticated algorithms for numerical optimization exist, in particular for smooth convex functions such as $E(W,b)$, stochastic gradient descent remains the method of choice when the number of training examples is too large to fit in memory, or in the setting where training examples arrive in a continuous stream. Even with relatively manageable dataset sizes, SGD can be particularly advantageous for non-convex loss functions (such as those explored in Benchmarking Results), where the stochasticity can allow the optimizer to escape shallow local minima [Bottou].

According to the SGD algorithm, the update on W is

$$W \leftarrow W - \mu\frac{1}{N'}\sum_i \left.\frac{\partial E(W,b,x,y)}{\partial W}\right|_{x=x^{(i)},y=y^{(i)}}, \qquad (4)$$

where $\mu = 0.1$ is the step size and N is the number of examples with which we will approximate the gradient (i.e. the number of rows of x). The update on b is likewise

$$b \leftarrow b - \mu\frac{1}{N'}\sum_i \left.\frac{\partial E(W,b,x,y)}{\partial b}\right|_{x=x^{(i)},y=y^{(i)}}. \qquad (5)$$

Implementing this minimization procedure in Theano involves the following four conceptual steps: (1) declaring symbolic variables, (2) using these variables to build a symbolic expression graph, (3) compiling Theano functions, and (4)

---

[4] Taking the mean in this fashion decouples the choice of the regularization coefficient and the stochastic gradient step size from the number of training examples.

calling said functions to perform numerical computations. The code listings in Figures 1 - 4 illustrate these steps with a working program that fits a logistic regression model to random data.

```
 1: import numpy
 2: import theano.tensor as T
 3: from theano import shared, function
 4:
 5: x = T.matrix()
 6: y = T.lvector()
 7: w = shared(numpy.random.randn(100))
 8: b = shared(numpy.zeros(()))
 9: print "Initial model:"
10: print w.get_value(), b.get_value()
```

Figure 1: *Logistic regression, part 1: declaring variables.*

The code in Figure 1 declares four symbolic variables x, y w, and b to represent the data and parameters of the model. Each tensor variable is strictly typed to include its data type, its number of dimensions, and the dimensions along which it may broadcast (like NumPy's broadcasting) in element-wise expressions. The variable x is a matrix of the default data type (float64), and y is a vector of type long (or int64). Each row of x will store an example $x^{(i)}$, and each element of y will store the corresponding label $y^{(i)}$. The number of examples to use at once represents a tradeoff between computational and statistical efficiency.

The shared() function creates *shared variables* for W and b and assigns them initial values. Shared variables behave much like other Theano variables, with the exception that they also have a persistent value. A shared variable's value is maintained throughout the execution of the program and can be accessed with .get_value() and .set_value(), as shown in line 10.

```
11: p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
12: xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
13: cost = xent.mean() + 0.01*(w**2).sum()
14: gw,gb = T.grad(cost, [w,b])
15: prediction = p_1 > 0.5
```

Figure 2: *Logistic regression, part 2: the computation graph.*

The code in Figure 2 specifies the computational graph required to perform stochastic gradient descent on the parameters of our cost function. Since Theano's interface shares much in common with that of NumPy, lines 11-15 should be self-explanatory for anyone familiar with that module. On line 11, we start by defining $P(Y = 1|x^{(i)}) = 1$ as the symbolic variable p_1. Notice that the matrix multiplication and element-wise exponential functions are simply called via the T.dot and T.exp functions, analogous to numpy.dot and numpy.exp. xent defines the cross-entropy loss function, which is then combined with the ℓ_2 penalty on line 13, to form the cost function of Eq (3) and denoted by cost.

Line 14 is crucial to our implementation of SGD, as it performs symbolic differentiation of the scalar-valued cost variable with respect to variables w and b. T.grad operates by iterating backwards over the expression graph, applying the chain rule of differentiation and building symbolic expressions for the gradients on w and b. As such, gw and gb are also

symbolic Theano variables, representing $\partial E/\partial W$ and $\partial E/\partial b$ respectively. Finally, line 15 defines the actual prediction (prediction) of the logistic regression by thresholding $P(Y = 1|x^{(i)})$.

```
16: predict = function(inputs=[x],
17:                     outputs=prediction)
18: train = function(
19:            inputs=[x,y],
20:            outputs=[prediction, xent],
21:            updates={w:w-0.1*gw, b:b-0.1*gb})
```

Figure 3: Logistic regression, part 3: compilation.

The code of Figure 3 creates the two functions required to train and test our logistic regression model. Theano functions are callable objects that compute zero or more *outputs* from values given for one or more symbolic *inputs*. For example, the predict function computes and returns the value of prediction for a given value of x. Parameters w and b are passed implicitly - all shared variables are available as inputs to all functions as a convenience to the user.

Line 18 (Figure 3) which creates the train function highlights two other important features of Theano functions: the potential for multiple outputs and updates. In our example, train computes both the prediction (prediction) of the classifier as well as the cross-entropy error function (xent). Computing both outputs together is computationally efficient since it allows for the reuse of intermediate computations, such as dot(x,w). The optional updates parameter enables functions to have side-effects on shared variables. The updates argument is a dictionary which specifies how shared variables should be updated after all other computation for the function takes place, just before the function returns. In our example, calling the train function will update the parameters w and b with new values as per the SGD algorithm.

```
22: N = 4
23: feats = 100
24: D = (numpy.random.randn(N, feats),
25:      numpy.random.randint(size=N,low=0, high=2))
26: training_steps = 10
27: for i in range(training_steps):
28:     pred, err = train(D[0], D[1])
29: print "Final model:",
30: print w.get_value(), b.get_value()
31: print "target values for D", D[1]
32: print "prediction on D", predict(D[0])
```

Figure 4: Logistic regression, part 4: computation.

Our example concludes (Figure 4) by using the functions train and predict to fit the logistic regression model. Our data D in this example is just four random vectors and labels. Repeatedly calling the train function (lines 27-28) fits our parameters to the data. Note that calling a Theano function is no different than calling a standard Python function: the graph transformations, optimizations, compilation and calling of efficient C-functions (whether targeted for the CPU or GPU) have all been done under the hood. The arguments and return values of these functions are NumPy ndarray objects that interoperate normally with other scientific Python libraries and tools.
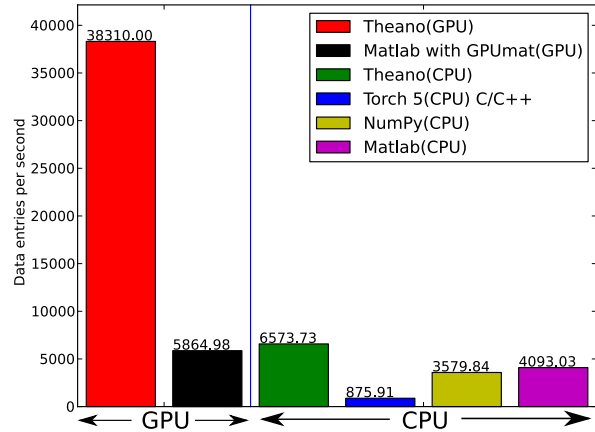
Figure 5: Fitting a multi-layer perceptron to simulated data with various implementations of stochastic gradient descent. These models have 784 inputs, 500 hidden units, a 10-way classification, and are trained 60 examples at a time.

Benchmarking Results

Theano was developed to simplify the implementation of complex high-performance machine learning algorithms. This section presents performance in two processor-intensive tasks from that domain: training a multi-layer perceptron (MLP) and training a convolutional network. We chose these architectures because of their popularity in the machine learning community and their different computational demands. Large matrix-matrix multiplications dominate in the MLP example and two-dimensional image convolutions with small kernels are the major bottleneck in a convolutional network. More information about these models and their associated learning algorithms is available from the Deep Learning Tutorials [DLT]. The implementations used in these benchmarks are available online [dlb].

CPU timing was carried out on an an Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz with 2 GB of RAM. All implementations were linked against the BLAS implemented in the Intel Math Kernel Library, version 10.2.4.032 and allowed to use only one thread. GPU timing was done on a GeForce GTX 285. CPU computations were done at double-precision, whereas GPU computations were done at single-precision.

Our first benchmark involves training a single layer MLP by stochastic gradient descent. Each implementation repeatedly carried out the following steps: (1) multiply 60 784-element input vectors by a 784×500 weight matrix, (2) apply an element-wise hyperbolic tangent operator (tanh) to the result, (3) multiply the result of the tanh operation by a 500×10 matrix, (4) classify the result using a multi-class generalization of logistic regression, (5) compute the gradient by performing similar calculations but in reverse, and finally (6) add the gradients to the parameters. This program stresses element-wise computations and the use of BLAS routines.

Figure 5 compares the number of examples processed per second across different implementations. We compared
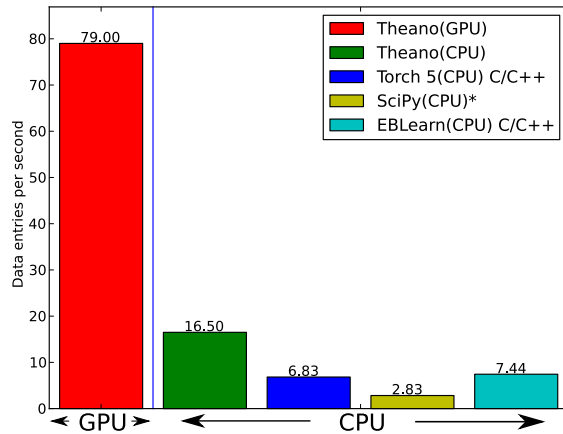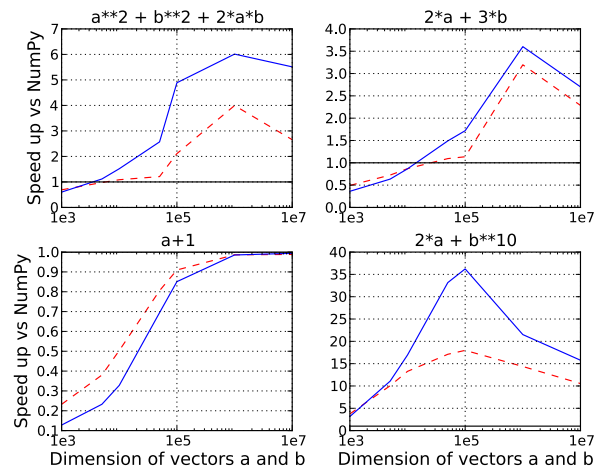
Figure 6: *Fitting a convolutional network using different software. The benchmark stresses convolutions of medium-sized (256 by 256) images with small (7 by 7) filters.*

Figure 7: *Speed comparison between NumPy, numexpr, and Theano for different sizes of input on four element-wise formulae. In each subplot, the solid blue line represents Theano, the dashed red line represent numexpr, and performance is plotted with respect to NumPy.*

Theano (revision #ec057beb6c) against NumPy 1.4.1, MAT-LAB 7.9.0.529, and Torch 5 (a machine learning library written in C/C++) [torch5] on the CPU and GPUMat 0.25 for MATLAB ([gpumat]) on the GPU.

When running on the CPU, Theano is 1.8x faster than NumPy, 1.6x faster than MATLAB, and 7.5x faster than Torch 5.[5] Theano's speed increases 5.8x on the GPU from the CPU, a total increase of 11x over NumPy (CPU) and 44x over Torch 5 (CPU). GPUmat brings about a speed increase of only 1.4x when switching to the GPU for the MATLAB implementation, far less than the 5.8x increase Theano achieves through CUDA specializations.

Because of the difficulty in implementing efficient convolutional networks, we only benchmark against known libraries that offer a pre-existing implementation. We compare against EBLearn [EBL] and Torch, two libraries written in C++. EBLearn was implemented by Yann LeCun's lab at NYU, who have done extensive research in convolutional networks. To put these results into perspective, we implemented approximately half (no gradient calculation) of the algorithm using SciPy's `signal.convolve2d` function. This benchmark uses convolutions of medium sized images (256×256) with small filters (7×7). Figure 6 compares the performance of Theano (both CPU and GPU) with that of competing implementations. On the CPU, Theano is 2.2x faster than EBLearn, its best competitor. This advantage is owed to the fact that Theano compiles more specialized convolution routines. Theano's speed increases 4.9x on the GPU from the CPU, a total of 10.7x over EBLearn (CPU). On the CPU, Theano is 5.8x faster than SciPy even though SciPy is doing only half the computations. This is because SciPy's convolution routine has not been optimized for this application.

We also compared Theano with numexpr and NumPy for evaluating element-wise expressions on the CPU (Figure 7).

---

[5] Torch was designed and implemented with flexibility in mind, not speed (Ronan Collobert, p.c.).

For small amounts of data, the extra function-call overhead of numexpr and Theano makes them slower. For larger amounts of data, and for more complicated expressions, Theano is fastest because it uses an implementation specialized for each expression.

What kinds of work does Theano support?

Theano's expression types cover much of the same functionality as NumPy, and include some of what can be found in SciPy. Table 1 lists some of the most-used expressions in Theano. More extensive reference documentation is available online [theano].

Theano's strong suit is its support for strided N-dimensional arrays of integers and floating point values. Signed and unsigned integers of all native bit widths are supported, as are both single-precision and double-precision floats. Single-precision and double-precision complex numbers are also supported, but less so - for example, gradients through several mathematical functions are not implemented. Roughly 90% of expressions for single-precision N-dimensional arrays have GPU implementations. Our goal is to provide GPU implementations for all expressions supported by Theano.

Random numbers are provided in two ways: via NumPy's random module, and via an internal generator from the MRG family [Ecu]. Theano's `RandomStreams` replicates the `numpy.random.RandomState` interface, and acts as a proxy to NumPy's random number generator and the various random distributions that use it. The `MRG_RandomStreams` class implements a different random number generation algorithm (called MRG31k3p) that maps naturally to GPU architectures. It is implemented for both the CPU and GPU so that programs can produce the same results on either architecture without sacrificing speed. The `MRG_RandomStreams` class

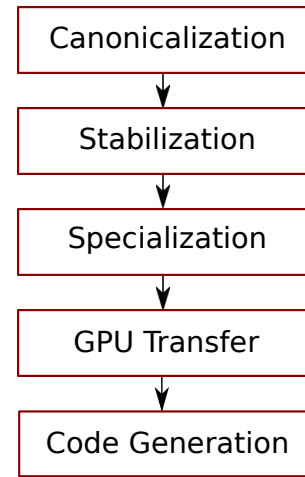| Operators	+, -, /, *, **, //, eq, neq, <, <=, >, >=, &, |, ^
Allocation	alloc, eye, [ones,zeros]_like, identity{_like}
Indexing*	basic slicing (see set_subtensor and inc_subtensor for slicing lvalues); limited support for advanced indexing
Mathematical Functions	exp, log, tan[h], cos[h], sin[h], real, imag, sqrt, floor, ceil, round, abs
Tensor Operations	all, any, mean, sum, min, max, var, prod, argmin, argmax, reshape, flatten, dimshuffle
Conditional	cond, switch
Looping	Scan
Linear Algebra	dot, outer, tensordot, diag, cholesky, inv, solve
Calculus*	grad
Signal Processing	conv2d, FFT, max_pool_2d
Random	RandomStreams, MRG_RandomStreams
Printing	Print
Sparse	compressed row/col storage, limited operator support, dot, transpose, conversion to/from dense
Machine Learning	sigmoid, softmax, multi-class hinge loss

TABLE I.: *Overview of Theano's core functionality. This list is not exhaustive, and is superseded by the online documentation. More details are given in text for items marked with an asterisk.* dimshuffle *is like* numpy.swapaxes.

Figure 8: *The compilation pipeline for functions compiled for GPU. Functions compiled for the CPU omit the GPU transfer step. v2*

offers a more limited selection of random number distributions than NumPy though: uniform, normal, and multinomial.

Sparse vectors and matrices are supported via SciPy's sparse module. Only compressed-row and compressed-column formats are supported by most expressions. There are expressions for packing and unpacking these sparse types, some operator support (e.g. scaling, negation), matrix transposition, and matrix multiplication with both sparse and dense matrices. Sparse expressions currently have no GPU equivalents.

There is also support in Theano for arbitrary Python objects. However, there are very few expressions that make use of that support because the compilation pipeline works on the basis of inferring properties of intermediate results. If an intermediate result can be an arbitrary Python object, very little can be inferred. Still, it is occasionally useful to have such objects in Theano graphs.

Theano has been developed to support machine learning

research, and that has motivated the inclusion of more specialized expression types such as the logistic sigmoid, the softmax function, and multi-class hinge loss.

Compilation by theano.function

What happens under the hood when creating a function? This section outlines, in broad strokes, the stages of the compilation pipeline. Prior to these stages, the expression graph is copied so that the compilation process does not change anything in the graph built by the user. As illustrated in Figure 8, the expression graph is subjected to several transformations: (1) canonicalization, (2) stabilization, (3) specialization, (4) optional GPU transfer, (5) code generation. There is some overlap between these transformations, but at a high level they have different objectives. (The interested reader should note that these transformations correspond roughly, but not exactly to the optimization objects that are implemented in the project source code.)

Canonicalization: The canonicalization transformation puts the user's expression graph into a standard form. For example, duplicate expressions are merged into a single expression. Two expressions are considered duplicates if they carry out the same operation and have the same inputs. Since Theano expressions are purely functional (i.e., cannot have side effects), these expressions must return the same value and thus it is safe to perform the operation once and reuse the result. The symbolic gradient mechanism often introduces redundancy, so this step is quite important. For another example, sub-expressions involving only multiplication and division are put into a standard fraction form (e.g. a / (((a * b) / c) / d) -> (a * c * d) / (a * b) -> (c * d) / (b)). Some useless calculations are eliminated in this phase, for instance cancelling out uses of the a term in the previous example, but also reducing exp(log(x)) to x, and computing outright the values of any expression whose inputs are fully known at compile time. Canonicalization simplifies and optimizes the graph to some extent, but its primary function is to collapse many different expressions into a single

normal form so that it is easier to recognize expression patterns in subsequent compilation stages.

Stabilization: The stabilization transformation improves the numerical stability of the computations implied by the expression graph. For instance, consider the function `log(1 + exp(x))`, which tends toward zero as $\lim_{x \to -\infty}$, and x as $\lim_{x \to -\infty}$. Due to limitations in the representation of double precision numbers, the computation as written yields infinity for `x > 709`. The stabilization phase replaces patterns like one with an expression that simply returns x when x is sufficiently large (using doubles, this is accurate beyond the least significant digit). It should be noted that this phase cannot guarantee the stability of computations. It helps in some cases, but the user is still advised to be wary of numerically problematic computations.

Specialization: The specialization transformation replaces expressions with faster ones. Expressions like `pow(x,2)` become `sqr(x)`. Theano also performs more elaborate specializations: for example, expressions involving scalar-multiplied matrix additions and multiplications may become BLAS General matrix multiply (GEMM) nodes and `reshape`, `transpose`, and `subtensor` expressions (which create copies by default) are replaced by constant-time versions that work by aliasing memory. Expressions subgraphs involving element-wise operations are fused together (as in numexpr) in order to avoid the creation and use of unnecessary temporary variables. For instance, if we denote the `a + b` operation on tensors as `map(+, a, b)`, then an expression such as `map(+, map(*, a, b), c)` would become `map(lambda ai,bi,ci: ai*bi+ci, a, b, c)`. If the user desires to use the GPU, expressions with corresponding GPU implementations are substituted in, and transfer expressions are introduced where needed. Specialization also introduces expressions that treat inputs as workspace buffers. Such expressions use less memory and make better use of hierarchical memory, but they must be used with care because they effectively destroy intermediate results. Many expressions (e.g. GEMM and all element-wise ones) have such equivalents. Reusing memory this way allows more computation to take place on GPUs, where memory is at a premium.

Moving Computation to the GPU: Each expression in Theano is associated with an implementation that runs on either the host (a host expression) or a GPU device (a GPU expression). The GPU-transfer transformation replaces host expressions with GPU expressions. The majority of host expression types have GPU equivalents and the proportion is always growing.

The heuristic that guides GPU allocation is simple: if any input or output of an expression resides on the GPU and the expression has a GPU equivalent, then the GPU equivalent is substituted in. Shared variables storing `float32` tensors default to GPU storage, and the expressions derived from them consequently default to using GPU implementations. It is possible to explicitly force any `float32` variable to reside on the GPU, so one can start the chain reaction of optimizations and use the GPU even in graphs with no shared variables. It is possible (though awkward, and discouraged) to specify exactly which computations to perform on the GPU by disabling the default GPU optimizations.

Tensors stored on the GPU use a special internal data type with an interface similar to the `ndarray`. This datatype fully supports strided tensors, and arbitrary numbers of dimensions. The support for strides means that several operations such as the transpose and simple slice indexing can be performed in constant time.

Code Generation: The code generation phase of the compilation process produces and loads dynamically-compiled Python modules with specialized implementations for the expressions in the computation graph. Not all expressions have C (technically C++) implementations, but many (roughly 80%) of Theano's expressions generate and compile C or CUDA code during `theano.function`. The majority of expressions that generate C code specialize the code based on the dtype, broadcasting pattern, and number of dimensions of their arguments. A few expressions, such as the small-filter convolution (`conv2d`), further specialize code based on the size the arguments will have.

Why is it so important to specialize C code in this way? Modern x86 architectures are relatively forgiving of code that does not make good use techniques such as loop unrolling and prefetching contiguous blocks of memory, and only the `conv2d` expression goes to any great length to generate many special case implementations for the CPU. By comparison, GPU architectures are much less forgiving of code that is not carefully specialized for the size and physical layout of function arguments. Consequently, the code generators for GPU expressions like `GpuSum`, `GpuElementwise`, and `GpuConv2d` generate a wider variety of implementations than their respective host expressions. With the current generation of graphics cards, the difference in speed between a naïve implementation and an optimal implementation of an expression as simple as matrix row summation can be an order of magnitude or more. The fact that Theano's GPU `ndarray`-like type supports strided tensors makes it even more important for the GPU code generators to support a variety of memory layouts. These compile-time specialized CUDA kernels are integral to Theano's GPU performance.

Limitations and Future Work

While most of the development effort has been directed at making Theano produce fast code, not as much attention has been paid to the optimization of the compilation process itself. At present, the compilation time tends to grow super-linearly with the size of the expression graph. Theano can deal with graphs up to a few thousand nodes, with compilation times typically on the order of seconds. Beyond that, it can be impractically slow, unless some of the more expensive optimizations are disabled, or pieces of the graph are compiledseparately.

A Theano function call also requires more overhead (on the order of microseconds) than a native Python function call. For this reason, Theano is suited to applications where functions correspond to expressions that are not too small (see Figure 5).

The set of types and operations that Theano provides continues to grow, but it does not cover all the function-

ality of NumPy and covers only a few features of SciPy. Wrapping functions from these and other libraries is often straightforward, but implementing their gradients or related graph transformations can be more difficult. Theano does not yet have expressions for sparse or dense matrix inversion, nor linear algebra decompositions, although work on these is underway outside of the Theano trunk. Support for complex numbers is also not as widely implemented or as well-tested as for integers and floating point numbers. NumPy arrays with non-numeric dtypes (strings, Unicode, Python objects) are not supported at present.

We expect to improve support for advanced indexing and linear algebra in the coming months. Documentation online describes how to add new operations and new graph transformations. There is currently an experimental GPU version of the scan operation, used for looping, and an experimental lazy-evaluation scheme for branching conditionals.

The library has been tuned towards expressions related to machine learning with neural networks, and it is not as well tested outside of this domain. Theano is not a powerful computer algebra system, and it is an important area of future work to improve its ability to recognize numerical instability in complicated element-wise expression graphs.

Debugging Theano functions can require non-standard techniques and Theano specific tools. The reason is two-fold: 1) definition of Theano expressions is separate from their execution, and 2) optimizations can introduce many changes to the computation graph. Theano thus provides separate execution modes for Theano functions, which allows for automated debugging and profiling. Debugging entails automated sanity checks, which ensure that all optimizations and graph transformations are safe (Theano compares the results before and after their application), as well as comparing the outputs of both C and Python implementations.

We plan to extend GPU support to the full range of C data types, but only float32 tensors are supported as of this writing. There is also no support for sparse vectors or matrices on the GPU, although algorithms from the CUSPARSE package should make it easy to add at least basic support for sparse GPU objects.

Conclusion

Theano is a mathematical expression compiler for Python that translates high level NumPy-like code into machine language for efficient CPU and GPU computation. Theano achieves good performance by minimizing the use of temporary variables, minimizing pressure on fast memory caches, making full use of `gemm` and `gemv` BLAS subroutines, and generating fast C code that is specialized to sizes and constants in the expression graph. Theano implementations of machine learning algorithms related to neural networks on one core of an E8500 CPU are up to 1.8 times faster than implementations in NumPy, 1.6 times faster than MATLAB, and 7.6 times faster than a related C++ library. Using a Nvidia GeForce GTX285 GPU, Theano is an additional 5.8 times faster. One of Theano's greatest strengths is its ability to generate custom-made CUDA kernels, which can not only significantly outperform CPU implementations but alternative GPU implementations as well.

REFERENCES

[theano] Theano. http://www.deeplearning.net/software/theano

[NumPy] T. E. Oliphant. "Python for Scientific Computing". *Computing in Science & Engineering* 9, 10 (2007).

[Bottou] L. Bottou. "Online Algorithms and Stochastic Approximations". In D. Saad, ed. *Online Learning and Neural Networks* (1998). Cambridge University Press, Cambridge, UK. Online: http://leon.bottou.org/papers/bottou-98x

[numexpr] D. Cooke *et al*. "numexpr". http://code.google.com/p/numexpr/

[Cython] S. Behnel, R. Bradshaw, and D. S. Seljebotn. "Cython: C-Extensions for Python". http://www.cython.org/

[scipy.weave] SciPy Weave module. http://docs.scipy.org/doc/scipy/reference/tutorial/weave.html

[Alted] F. Alted. "Why Modern CPUs Are Starving And What Can Be Done About It". *Computing in Science and Engineering* 12(2):68-71, 2010.

[SymPy] SymPy Development Team. "SymPy: Python Library for Symbolic Mathematics". http://www.sympy.org/

[BLAS] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. "Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms". *ACM Trans. Math. Soft.*, 16:18-28, 1990. http://www.netlib.org/blas

[LAPACK] E. Anderson *et al*. "LAPACK Users' Guide, Third Edition". http://www.netlib.org/lapack/lug/index.html

[DLT] Deep Learning Tutorials. http://deeplearning.net/tutorial/

[dlb] Benchmarking code: http://github.com/pascanur/DeepLearningBenchmarks

[torch5] R. Collobert. "Torch 5". http://torch5.sourceforge.net

[EBL] "EBLearn: Energy Based Learning, a C++ Machine Learning Library". http://eblearn.sourceforge.net/

[gpumat] "GPUmat: GPU toolbox for MATLAB". http://gp-you.org

[Ecu] P. L'Ecuyer, F. Blouin, and R. Couture. "A Search for Good Multiple Recursive Generators". *ACM Transactions on Modeling and Computer Simulation*, 3:87-98, 1993.